A Thread Synchronization Model for SIP Servlet Containers

Yi Huang[a] Eric Cheung[b] Laura K. Dillon[a] R. E. Kurt Stirewalt[a]

[a]Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, US
{huangyi7,ldillon,stire}@cse.msu.edu

[b]AT&T Research Labs, Inc.
180 Park Avenue
Florham, NJ 07932, US
cheung@research.att.com

ABSTRACT

Multi-threaded SIP servlet containers pose difficult synchronization problems to application developers. On the one hand, if a container automatically locks resources for servlets according to some fixed protocol, this protocol is not likely to be appropriate for all applications. It may degrade performance or even introduce deadlock in some applications. On the other hand, leaving application programmers to code synchronization—while flexible—is prone to error. The logic to both infer the resources a servlet requires and then to lock the required resources is complex and cross-cutting. This "synchronization logic" can easily obscure an application's "business logic." Interleaving synchronization code with business code makes an application difficult to maintain and extend.

In this paper, we elaborate key dimensions of the thread synchronization problem for SIP servlet containers. We further propose a novel synchronization model for SIP servlet containers, which addresses these dimensions in a more comprehensive fashion than any existing model that we know of. Our synchronization model is highly flexible. It introduces abstractions to promote correctness, maintainabilty, and extensibility. We also describe a reference framework that implements these abstractions. To illustrate the benefits of our model, we describe a representative SIP application developed using this framework.

1. INTRODUCTION

A typical Voice-over-IP (VoIP) application consists of a set of *servlets* and is deployed to a *servlet container*. The container automates many low level details of the management of *sessions*. An *application session* represents ongoing instances of an application and may consist of multiple *protocol sessions*—e.g., Session Initiation Protocol (SIP) sessions and HyperText Transfer Protocol (HTTP) sessions—each representing an independent connection between autonomous agents. Because agents send messages to initiate, terminate, or otherwise modify sessions independently, the container cannot predict when or in what order messages will arrive. Thus, to ensure responsiveness, the SIP servlet standard (JSR 289 [18]) prescribes that, upon receiving a SIP message, a SIP servlet container determines the application and the servlet within that application to which the message should be routed as well as the application and protocol sessions associated with the message, and that it then dispatches the message to a dedicated thread to execute the servlet, thereby "processing" the message.

When writing applications for such containers, synchronization is an important issue that complicates application development. The need to synchronize threads executing within a container arises because multiple threads processing different messages may attempt to concurrently access the same session data. Additionally, a thread may create and use data that persists beyond the thread's lifetime and, consequently, that may be accessed by multiple threads. Synchronization affects not only throughput, but also correctness, as synchronization errors can result in data races and deadlock [10].

Ideally, a SIP container would automatically synchronize threads so that each thread has exclusive access to the shared objects needed by the servlet that it executes. However, the container cannot know *a priori* what sessions a thread will need to access in processing a message. Because servlets are stateless, the thread may have to inspect some session data to discover other sessions and data that it needs to access in processing a message. Moreover, the sessions and data it needs may change dynamically as the servlet executes. Serializing the processing of all messages bound to the same application session might seem like a safe synchronization model for a container to implement. In fact, some existing commercial containers adopt this synchronization model. However, this simple model is at the same time overly pessimistic and insufficient. It is overly pessimistic because a single application session may contain many protocol sessions, and the sessions that different threads need to access in processing messages bound for the same application session are often disjoint. It is insufficient because threads may need to access additional shared resources (i.e., in addition to the session data) to process some messages. These considerations imply that synchronization of threads cannot be entirely relegated to a container for general VoIP applications.

Many application programmers prefer to program synchronization explicitly—for instance, using monitors and mutex locks—because doing so provides them control and flexibility. However, synchronization code is notorious for being low level, as it is complex and error prone [14]. Moreover, when interleaved with business code, it obscures the business logic. As a result, applications become brittle—breaking easily under extension and maintenance. Additionally, because servlets are stateless, SIP threads tend to incrementally lock resources. Incremental locking, in turn, can lead to deadlocks.

To better equip programmers for developing reliable and efficient SIP applications, we propose a high-level synchronization model for SIP servlet containers. Our model represents the business logic of an application as a (potentially infinite) state automaton. It introduces a *synchronization contract* for representing an application's resource needs at a suitably abstract level. Intuitively, the synchronization contract specifies the resources a thread should release and acquire as the thread takes transitions.

This manner of specification makes it possible to implement the synchronization logic for a SIP application in a middleware framework.[1] Briefly, the programmer does not mix business code with the code to infer the resources a thread needs and to lock them. Instead, she embeds calls to the middleware to effect transitions between states. When invoked, the middleware consults the application's synchronization contract and dynamically infers the resources the thread should release and then lock prior to taking the transition. The middleware then proceeds to release the former and *negotiate* on behalf of the thread to acquire the latter. During this negotiation, the middleware will block the thread if another thread has the lock on a resource that this thread needs. It might also block the thread and yield to another thread contending for a resource that it needs in order to be fair or avoid deadlock—thus, the term "negotiation". In any case, the middleware returns control to the application only once the needed resources are acquired.

Handling synchronization in this manner offers several important benefits over existing mechanisms for synchronizing SIP threads.

Flexibility: Rather than prescribing a fixed synchronization protocol, our model allows the programmer to provide a synchronization contract specifying the resource needs of an application. Using the contract, the framework tailors a synchronization protocol to the particular application. Additionally, our middleware framework allows swapping of different synchronization protocols and contract languages.

Traceability: Encapsulating the low level synchronization code in a framework and embedding calls announcing state transitions in the business code makes the business code more easily traceable to a conventional state diagram that describes the business logic. In turn, this traceability makes the code easier to comprehend, maintain and extend.

[1]or in a SIP servlet container itself. For expediency, our prototype uses a framework.

Correctness: Better traceability also simplifies reasoning about correctness of the business code. Moreover, concurrent programming experts can instantiate and validate the middleware framework, which any SIP application programmer can then reliably use. In particular, the instantiated framework can incorporate heuristics for avoiding and detecting data races and deadlocks, for enforcing fairness or priorities, and so on.

We created the *Synchronization Negotiation Framework for SIP Servlets (SNeF4SS)* for initial testing that our model achieves these goals. SNeF4SS defines a simple API for realizing our model, and provides three alternative implementations of the API, each of which implements a different synchronization protocol and displays different efficiency tradeoffs in different scenarios of use [11]. The framework permits swapping synchronization protocols so that a programmer can experiment with different protocols to determine which is best suited for a particular application and execution environment. SNeF4SS also provides a simple contract language. As a pilot study, we developed a dating service application using SNeF4SS.

This paper builds on our prior work, in which we documented two deadlock scenarios that can occur in applications deployed to a popular SIP servlet container, and proposed an approach to avoiding those deadlock scenarios [10].

In sum, this paper makes the following contributions.

- It elaborates the general thread synchronization problem in SIP servlet containers, examining implications of container architectures on this problem (**Section 2.4**).
- It introduces a state-based notation for declaring the evolving synchronization needs of a SIP application (**Section 3**).
- It describes SNeF4SS—a generic thread synchronization middleware framework, which promotes correctness, maintainability and extensibility and which works with a SIP servlet container right off the shelf (**Section 4**).
- It defines a high level, formal model for synchronizing SIP threads that provides a semantic foundation for this framework (**Section 5**).

2. BACKGROUND

Applications deployed to SIP servlet containers are inherently multi-threaded, and these threads must access and update shared resources, such as persistent session objects. To protect these shared resources from data corruption, threads must synchronize according to well-defined synchronization protocols. In the SIP domain, a synchronization protocol implements logic (1) to dynamically infer the resource needs of threads and (2) to release and acquire resources in a manner that guarantees mutually-exclusive access to a resource or collection of resources. We refer to a protocol for (1) as a *resource inference protocol* and a protocol for (2) as a *locking protocol.*[2] The amount of contention incurred by any given

[2]Lock-free approaches to synchronization, such as *transactional memory* [13], are largely inappropriate in the VoIP

synchronization protocol is sensitive to patterns of resource usage within the environment.

In this section, we describe the conceptual architecture of typical SIP servlet containers, focusing on the granularity of concurrency and the typical patterns by which concurrent threads access shared resources (**Section 2.1**). To illustrate the key concepts, we describe a simple, but representative application (**Section 2.2**). We also describe the resource sharing characteristics of this application (**Section 2.3**) as well as the implied synchronization issues and their common solutions (**Section 2.4**).

2.1 SIP servlet containers

A SIP servlet container provides a runtime environment for SIP applications. To be deployed to a container, a SIP application must define one or more servlets, which encapsulate the logic for processing messages sent to the application by user agents. Servlets are required to be stateless, so that they can concurrently host multiple threads. Thus, a container needs just one instance of each servlet of a deployed application. To enable the use of singleton servlets, the container encapsulates the context needed to process semantically related messages into sessions.

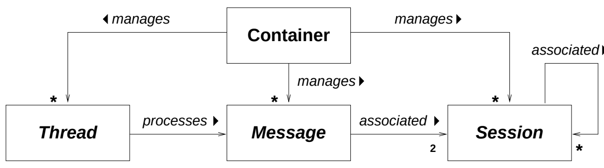

Figure 1: Class diagram of relationships between a container and the key types of resources it manages

Figure 1 depicts the relationships between a SIP servlet container and the key types of resources that it manages. We reify a container as an instance of a hypothetical class `Container`. At a minimum, a container manages threads, messages, and sessions, which we reify as instances of classes `Thread`, `Message`, and `Session`, respectively. In practice, different types of threads, messages and sessions are reified as instances of different concrete descendants of classes `Thread`, `Message` and `Session`, respectively. For simplicity of exposition, we refer to instances of these abstract classes when a distinction between specific concrete subclasses is irrelevant to our argument. Thus, for example, a message may be a SIP request message, a SIP response message, or even a message of another protocol, such as an HTTP message.

A thread processes exactly one message. In contrast, a message is associated with two sessions: a *protocol session* and an *application session*. A protocol session encapsulates the context needed to process messages pertaining to the same protocol connection. For example, the INVITE message that initiates a SIP call and a BYE message that terminates the same call belong to the same SIP session. An application session encapsulates the context of an application instance. As many SIP applications (e.g., B2BUA applications) handle multiple protocol connections, an application session often has multiple protocol sessions associated with it. For brevity, we refer to the application session and the protocol session that a message is associated with, collectively, as the message's *associated sessions*. The container provides an API for navigating and querying a message's associated sessions.

domain. Many actions necessary for processing messages cannot be revoked—for example, transmitting newly created messages over the network. Also, buffering newly created messages and postponing their transmission until the thread successfully commits may impede application responsiveness.

A container executes in a dedicated thread, distinct from the threads it manages. It listens for incoming messages on the network and for messages generated by the threads it manages. Upon receiving a message bound for a deployed application, it determines the servlet to route the message to. Additionally, if the message is not associated with a preexisting protocol session and/or application session, it creates the appropriate sessions; otherwise, it retrieves the message's associated sessions. The container then dispatches a thread to process the message, passing that thread the message, the associated servlet, and sessions. After dispatching the thread, the container returns to listening for other messages.

To process a message, a thread invokes a special method [3] on the servlet. The thread terminates on return from this method. While processing the message (in this special `service` method), the thread may create and send new messages using any protocols supported by the container.

Besides sessions and messages, a thread may create and use data that may be accessed concurrently by other threads. For instance, a thread may create a list or any other data object and store it to an attribute of a protocol or application session. Subsequently, this data object will be accessible to any thread processing a message associated with this same session. It may also be accessible to other threads through attributes of the sessions associated with the messages they are processing. The example in **Section 2.2** illustrates this situation.

2.2 Dating service example

For purposes of illustration, we present a design for a SIP application that implements a simple dating service. Briefly, a user calls the service in order to be connected with another user, who has also called the service for the same purpose. Neither user knows the identity of the other in advance of being connected. Once connected, the users may exchange information to arrange a "date."

Users of the dating service establish connections with one another by dialing into a central service using SIP phones. When a user U_1 dials in, his phone creates and sends an INVITE message m_1 to the container where the dating service is deployed. Because m_1 is an initial INVITE message, the container creates a new SIP session for m_1 and determines both the application session to associate with it and the servlet to which it should be routed for processing.[4] The

[3]the `service` method, which in turn invokes either the servlet's `doRequest` or `doResponse` method and a message-specific method—`doInvite` for an INVITE method, etc.

[4]The container chooses the servlet based on, for example, information contained in m_1's header.

container then creates a new thread, which invokes a method on this servlet to process m_1.

INVITE messages are processed by first *registering* the user (e.g., U_1) on a centralized list of available users who wish to be paired and then entering a *choosing* phase, during which the service attempts to locate a compatible user. In an actual dating service application, choosing requires some degree of interaction between the service and one or more of registered users. For instance, choosing may involve matching based on user profiles and preferences followed by a sequence of interactions whereby the users refine selection criteria and winnow down a set of candidate matches. Choosing may also not succeed for an observably long period of time. This phase must, therefore, provide some timeout functionality so the user agent who sent the INVITE message is not kept waiting infinately.

For brevity, the remainder of this paper focuses only on the processing required to handle initial INVITE messages. Having registered U_1, the thread inspects the list to determine whether another user, call her U_2, previously dialed into the service and is available to be paired with U_1. If such a U_2 exists and no further interaction with U_1 is required, the thread creates OK messages to send to the phones of U_1 and U_2 and then updates the list to reflect that U_2 is no longer available for pairing. At this point, the processing of m_1 is complete, the servlet invocation returns, and the thread is destroyed.

The patterns of resource access that are inherent to many SIP applications require complex synchronization protocols that allow threads to gain mutually-exclusive access to sets of resources so as to prevent high-level data races [1]. To see why, we now describe how these patterns manifest themselves in our dating service application by considering how threads created to process messages from distinct user agents share resources.

2.3 Resource sharing in this application

Perhaps the most important design decision pertaining to resource sharing involves how to store, access and update the set of registered, but as yet unpaired, users. To maximize the potential for finding a compatible partner, the service must store information about such users in a location that can be accessed by threads processing messages on behalf of any user agent. A convenient means for implementing centralized resources is to have the container assign the *same application session*[5] to messages arriving from multiple user agents. While convenient, contention for exclusive access to this resource could quickly become a synchronization bottleneck. To reduce contention, threads should release this resource as soon as possible rather than hold it for the duration of multi-step operations. In addition, some aspects of the choosing phase, such as interactions between the service and individual user agents to refine a selection, could operate on a local copy of some subset of this centralized list and thus need not contend for it during selection refinement.

Another decision that affects resource sharing concerns how to represent the status of each user and how to implement a status change due to successful pairing. Consider what must happen when a thread operating on behalf of some user U_1 wishes to commit to pairing with a peer user U_2. Another thread working on behalf of another user (e.g., U_3) may also have decided to commit to pairing with U_2 at about this same time. We could use the list of available users to record the availability status of each user; however, because this list is linked to the unique application session, doing so would require users such as U_1 to lock the list to change their status and the status of U_2 and users such as U_3 to lock the list to test if U_2 is still available before committing. Thus, given our decision to minimize contention for this centralized resource, we instead employ a design that stores the status of each individual user connection in the SIP session that represents that connection. The *pairing* of U_1 and U_2 is then accomplished by modifying the SIP sessions associated with U_1 and U_2 and then sending messages to notify U_1 and U_2 of the decision.

This decision regarding how to implement and update user status has several important consequences: First, the thread working on behalf of U_1 to issue the OK messages to the phones of U_1 and U_2 must first access the SIP sessions for both of these user agents. When it has acquired both sessions and determined that $U2$ has not already been paired with another peer user agent, it can safely record changes in their status by, for example, caching a reference to U_1's SIP session in U_2's SIP session and vice versa. Second, because changes in status are made at the level of SIP sessions, the centralized list will need to be updated once a change in status is made. This design reduces contention for the centralized list during pairing, because only one of perhaps many suitors will need to lock the list in order to commit to a pairing. All other suitors will find that they are too late to commit to U_2 when, after locking U_2's SIP session, they discover her status has changed to unavailable. While many suitors may block contending for this SIP session, they are not contending for the centralized application session and thus not blocking the progress of other parties that are not interested in U_2. Finally, another consequence of this design is that the centralized list will become stale for short periods of time between when the SIP sessions of a couple are updated but before the master list can be re-acquired and updated.

2.4 Synchronization issues

With these design decisions in mind, we now examine the requirements on a locking protocol that guarantees mutual exclusion while avoiding deadlock and starvation. In our scenario involving the processing of message m_1 on behalf of user U_1, processing begins by locking the application session so as to register U_1 and the SIP session associated with U_1. The SIP session associated with U_1 may then be released before the thread enters the choosing phase, thereby allowing other users to commit to partnering with U_1; however, the thread processing m_1 should retain the lock on the application session in order to access the list of available users. In the choosing phase, the thread processing m_1 should release

[5]The *Session Key Based Targeting Mechanism* defined by JSR 289 can be used to achieve this. The alternative is to create a new application session for each newly created SIP session. This implementation results in deadlock under the synchronization model employed by one container currently in use. In [10], we showed how we proposed to prevent this deadlock.

the application session immediately after finding a compatible user U_2. After releasing the application session, this thread should then lock the SIP sessions associated with U_1 and U_2 to modify their status appropriately and create and send out OK messages to notify U_1 and U_2 of the decision. This thread must then re-acquire the application session to mark U_1 and U_2 as unavailable.

This scenario illustrates two requirements on the locking protocol. First, it must be able to acquire multiple resources, e.g., "the application session AND the SIP session associated with U_1," "the SIP session associated with U_1 AND the SIP session associated with U_2," etc., without incurring deadlock. These resources may even include objects other than sessions. Second, the resources to lock must be known *a priori*. However, it is challenging to achieve this. For instance, the thread processing m_1 has no way of knowing it needs to acquire the SIP session associated with U_2 until U_2 is identified in the choosing phase. Even in an example as simple as this dating application, synchronization requirements mandate a complex locking protocol. Before presenting our solution, we now briefly close this section by discussing why two common approaches to locking protocols—manual design and container-managed synchronization—are insufficient.

A programmer may manually implement a locking protocol by writing code that explicitly acquires and releases resources. While powerful, this approach often fails in practice because correct and efficient protocols are very difficult to design, especially when (1) synchronization involves the atomic acquisition of multiple resources and (2) not all needed resources are known *a priori*. For sake of correctness, programmers often incorporate general deadlock-free strategies for acquiring multiple resources. Interleaving these strategies with in-progress resource inference further complicates the design. Two such strategies that are commonly used include *gatekeeper strategy* and *resource numbering strategy*. These strategies are described in detail in [10, 11]. However, depending on the threading characteristics of an application, these strategies may introduce unnecessary thread contention and unfair thread scheduling. Moreover, the optimal choice of strategy may become apparent only during the production stage, once the strategy has been fully integrated into the application code [11].

Another approach involves container-managed synchronization policies, whereby the container instructs a thread to acquire a set of resources prior to processing a message and to hold these resources until message processing is complete. Such policies are predominant in existing container implementations. However, as our dating service example illustrates, the container cannot determine the exact set of resources merely by examining the message. Under these circumstances, the container ends up acquiring and holding only a subset of the resources a thread needs to process a message. Because a thread's resource needs change as it processes a message, these lock-and-hold strategies are prone to inefficiency. Worse, in order to acquire the additional resources, a programmer must manually implement a custom synchronization protocol. In [10], we described how custom protocols operating alongside container-managed protocols can lead to unrecoverable deadlocks and described two cases that had occurred in practice.

To recap, the evolving synchronization needs of SIP applications complicate the design of the locking protocols needed to protect access to shared resources. Our contribution uses declarative specifications of evolving resource needs to guide the instantiation of a "synchronization middleware" framework to yield a deadlock-avoiding locking protocol that guarantees mutual exclusive access to resources as prescribed by the declarative specification.

3. SPECIFYING LOCKING PROTOCOLS

We now introduce the first component of our two-part solution, namely a state-based notation for declaring the evolving synchronization needs of a SIP application. The notation allows application designers to specify the behavior of message-processing threads using a variant of the UML StateChart notation where the states represent meaningful abstract states in the business logic. We extend this notation by allowing each state to be adorned with sets of expressions that identify the resources the application requires while executing within that state.

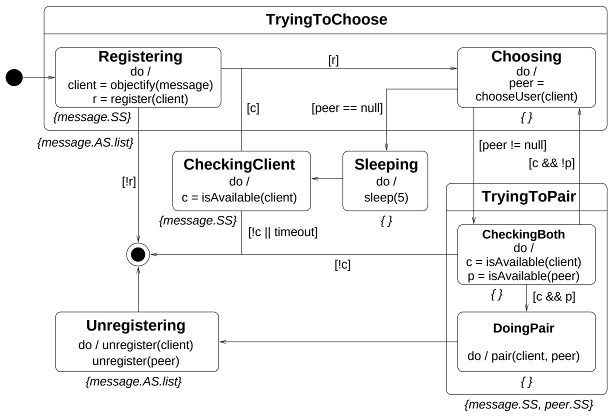

Figure 2: A state diagram of the dating service application

Figure 2 depicts a portion of our model of the dating service application. This portion focuses on the synchronization needs of threads that are spawned to process initial INVITE messages. Additional components of this model provide similar diagrams (not shown) that depict the evolving synchronization needs of threads that are spawned to process other messages. The adornments appear as sets of *navigation expressions*, listed in curly braces below the state to which they apply. A navigation expression specifies how the thread navigates the sessions associated with a message to locate a resource. We reserve the identifier `message` to denote the message dispatched to a thread and the attributes `SS` and `AS` to denote the SIP and application sessions associated with a given message. Data associated with session objects is specified by navigating through an attribute of the session object. Suppose, for instance, that `list` names an attribute of an application session object. Then the navigation expression `message.AS.list` denotes the object retrieved by navigating from the application session associated with the message dispatched to this thread through this `list` attribute.

Threads modeled by this diagram begin in the `Registering` state, as indicated by the UML start-state indicator (i.e., the black spot with an outgoing arrow). This state encapsulates a *do activity*, i.e., a computational process that persists in time. Here, the activity comprises two sub-activities. First, the message is *objectified* to produce an object (called `client`) that represents the user agent[6] that sent the message. This user agent, which corresponds to U_1 in our example from **Section 2.2**, is registered so that other users can know that this user is available for pairing. Registration involves adding `client` to a list that records users who have dialed in and are not yet paired. This list is linked to the application session through the session's `list` attribute.

Because this list of users is a shared resource, a thread working in the `Registering` state requires exclusive access to it while in that state. In fact, this list is also required in the `Choosing` state, which the thread enters after successfully registering `client`. Successful registration is determined by a boolean return value `r`, which is used to guard the transition from `Registering` to `Choosing`. To record this thread's need to hold exclusive access to the list while in these two states, our model provides the super-state `TryingToChoose` which we adorn with the navigation expression `message.AS.list`. Of course, to navigate to the `list` attribute, the thread must also access the application session itself. Super-state adornments are inherited by all sub-states nested within. Thus, when executing within the `Registering` state, a thread requires exclusive access to the list of registered users and the SIP and SIP application sessions associated with the message being processed.

Continuing with our example, a thread in the `Choosing` state performs an activity represented here as a call to the function `chooseUser`, which returns a user agent (`peer`). Next, it checks that the client and the chosen peer are both available (in `CheckingBoth`) and pairs them (in `DoingPair`), thereby creating and sending the OK messages. Finally, the thread "unregisters" `client` and `peer` (i.e., removes these user agents from the list stored in the application session's `list` attribute) and terminates (in the sink state).

Other paths through the state diagram correspond to other usage scenarios. For example, if `client` is the only user agent registered when a thread attempts to choose a peer (i.e., in `Choosing`, the call to `chooseUser` returns `null`), then the thread sleeps for at least five seconds. After this delay, the thread checks that `client` is still available (in `CheckingClient`). If unavailable (i.e., `c` is false), the thread terminates. This case occurs if some other thread selects and successfully pairs with `client` while this thread is sleeping. "Checking" states, like `CheckingClient` and `CheckingBoth`, are characteristic of applications deployed to SIP containers because of the asynchrony inherent in the container architecture.

Notice that our notation allows resource needs to be adorned with super-states. Such a specification is not equivalent to replicating the super-state needs in the adornments of all nested sub-states because the former indicates that resources should be held continuously during sub-state transitions, but the latter does not. To understand the implications of these semantics, notice that both states `CheckingBoth` and `DoingPair` require exclusive access to `client` and `peer` SIP sessions. In addition, state `DoingPair` is designed to assume that the condition checked in state `CheckingBoth` continues to hold throughout the transition from `CheckingBoth` to `DoingPair`. For this assumption to hold, other threads must not be allowed to access and/or modify the SIP sessions associated with either `client` or `peer`, as doing so may change their availability and thus invalidate the check performed in `CheckingBoth`.[7] Had we designed the system without the use of `TryingToPair`, instead copying the adornment of `TryingToPair` into the adornments of its sub-states, then the resources would be released upon exit of `CheckingBoth` and then re-acquired on entry to `DoingPair`. A system implemented according to such a specification would allow another thread to acquire and modify either or both of these SIP sessions during this thread's transition from `CheckingBoth` to `DoingPair`.

To summarize, our notation allows for the unambiguous specification of sets of resources needed by threads in each distinct abstract state of the business logic implemented by that thread. Because these needs may change when a thread transitions from one state to another, the model captures the evolution of these needs throughout the thread's lifetime. Finally, the actual resources that are needed are automatically inferred according to the navigation expressions.

4. SNeF4SS

We advocate encoding the evolving synchronization needs of a SIP application into explicit *synchronization contracts*, which reify the state-based specifications of evolving resource needs described in **Section 3**. To assess the efficacy of this approach, including whether such a specification could be expressed separately and then integrated cleanly with the business logic of a SIP application, we developed the *Synchronization Negotiation Framework for SIP Servlets* (SNeF-4SS). For expediency, we implemented SNeF4SS as a middleware framework, rather than modifying an existing container or creating our own. We tested this framework on the dating service application using the Sailfin container.[8] For brevity, we present here a simplified version of this application.

SNeF4SS encapsulates synchronization code in special objects, called *negotiators*, which consult an application's synchronization contract to infer the resources needed by a given thread as it cycles through its abstract business-logic states. SNeF4SS provides an API for creating negotiators and associating them with messages and for notifying negotiators as a thread transitions among these states (**Section 4.1**). A *framework programmer* instantiates SNeF4SS with a specific contract language and for negotiators that negotiate contracts written in that language. Our reference implementation provides a simple contract language (**Section 4.2**) and implementations for three types of negotiators, each implementing a different locking protocol (**Section 4.3**). Thus, framework programmers can instantiate SNeF4SS with implementations from our toolkit or write their own. In contrast to a framework programmer, an application programmer writes the servlet code expressing the business logic (**Section 4.4**). The servlet code embeds calls

[6]Here, the agent is assumed to be a SIP phone.

[7]This is an example of a *time of check to time of use* error.

[8]Sailfin: https://sailfin.dev.java.net

on a negotiator to effect transitions between thread states.

4.1 Framework overview

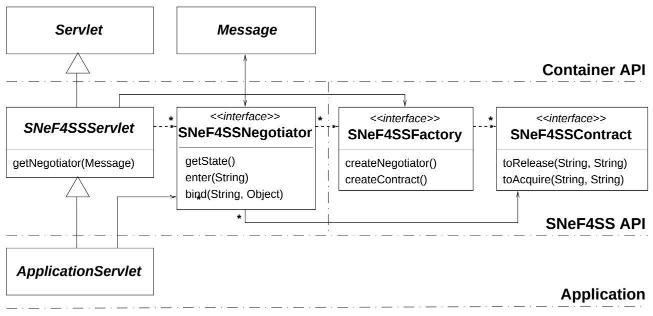

Figure 3: Class diagram of SNeF4SS API

The class diagram in **Figure 3** describes the SNeF4SS API. A framework programmer supplies implementations for all interface classes, marked by the `<<interface>>` stereotype, in the SNeF4SS layer (middle layer). An application programmer uses only those SNeF4SS-layer classes that are on the left of the dashed vertical line.

The SNeF4SS layer provides a variant for each concrete servlet class in the container API layer. For brevity, we represent all of these container-layer classes, including `SipServlet` and `HttpServlet`, by the abstract class `Servlet` and all of the SNeF4SS layer variants by the abstract class `SNeF4SSServlet`.[9] Each `SNeF4SSServlet` class wraps various methods of its base class. For instance, class `SNeF4SSSipServlet` wraps a method that the container invokes when it deploys the application to initialize the servlet context, extending this method to bind the servlet to the (singleton) `SNeF4SSFactory` object and a `SNeF4SSContract` object.[10] Class `SNeF4SSSipServlet` also wraps methods that a SIP thread automatically invokes upon starting up. Among other things, it extends each such method to create a negotiator—i.e., an object of type `SNeF4SSNegotiator`—and binds the message being processed to this negotiator.

In SNeF4SS, the application programmer provides a contract specification in the application's deployment descriptor. SNeF4SS loads this specification into a `SNeF4SSContract` object upon deployment of the application, as indicated above. The contract specification designates a set of *thread states* and a mapping from thread states to sets of navigation expressions to use for inferring the resources a thread needs to hold while in a state. It also specifies the initial thread state for a thread.

A negotiator maintains the state of the thread processing a message and maintains bindings for the names in navigation expressions. Briefly, a thread invokes the `getState` method on its negotiator to learn its current state, the `bind` method to bind a name to an actual resource, and the `enter` method to enter a new thread state. In the `enter` method, the negotiator consults the contract to infer what resources to release and then acquire. In case other threads hold some of the resources to be acquired, the negotiator blocks the thread until the resources become available. Thus, when executing in a state, a thread can safely assume it holds the resources designated by the specification of the application's contract.

[9]Names of abstract classes are set in an oblique font.

[10]Using a factory to create negotiators and contracts insulates the application code from details of the specific contract language and implementation. This allows the framework programmer to swap out different contract languages and implementations without requiring changes by the application programmer. Further discussion of this capability is beyond the scope of this paper.

4.2 Contracts

We implemented a simple XML-based language for specifying contracts. **Figure 4** shows part of a contract specification. Line numbers are for reference purposes; they are not part of the specification. The specification declares the thread state `TryingToChoose` (lines 3–17) as well as its nested states `Registering` (lines 5–10) and `Choosing` (lines 11–13) from **Figure 2**. It also declares a rule for starting a thread in `Registering`.

```
 1 <contract>
 2  <thread_state_decl>
 3   <thread_state>
 4     <state_name>TryingToChoose</state_name>
 5     <thread_state>
 6       <state_name>Registering</state_name>
 7       <state_needs>
 8         <need type="ps">message.SS</need>
 9       </state_needs>
10     </thread_state>
11     <thread_state>
12       <state_name>Choosing</state_name>
13     </thread_state>
14     <state_needs>
15       <need type="object">message.AS.list</need>
16     </state_needs>
17   </thread_state>
18  </thread_state_decl>
19   ...
20  <initial_thread_state_rule_decl>
21   <initial_state_rule>
22     <condition>
23       <equal>
24         <var>message.param.method</var>
25         <value>INVITE</value>
26       </equal>
27     </condition>
28     <target_state>
29       <state_name>Registering</state_name>
30     </target_state>
31   </initial_state_rule>
32    ...
33  </initial_thread_state_rule_decl>
34 </contract>
```

Figure 4: Example synchronization contract

A declaration for a super state embeds declarations for the states nested directly within it. Thus, `TryingToChoose` contains `Registering` and `Choosing`. A *state-needs* section in the declaration for a state lists one or more navigation expressions—for use in inferring the resources a thread needs to hold when executing in the state. The reserved identifiers `message`, `SS`, and `AS` denote, respectively, the message the thread is processing, an attribute referencing the SIP session associated with this message, and an attribute referencing the application session associated with this message. The parameter `type` indicates the type of a resource: `ps` signifies a protocol session, `as` an application session, and `object` a non-session object. Other reserved identifiers are not shown here and thus are omitted for brevity.

For example, the state-needs section (lines 14–16) for `TryingToChoose` indicates that, while in this state, a thread needs exclusive access to the object referenced by the `list` attribute of the message's application session. By default, we assume it also needs to hold the sessions along the access path to a needed resource. Thus, in this example, the thread also needs to hold the message's application session while in `TryingToChoose`. Moreover, in state `Registering`, a thread also needs the message's SIP session (lines 7–9), whereas, in `Choosing`, it does not need any additional resources besides the list and the application session. To enter `Registering`, therefore, a thread should acquire both the application session, the SIP session associated with the message and also the list of registered users. To transition from `Registering` to `Choosing`, a thread should release the SIP session, but not the list or application session. Finally, in taking a transition out of `TryingToChoose`, a thread should release all the resources it currently holds.

An initial state rule specifies a state and a condition for a thread to start executing in that state. The initial state rule (lines 21–31) of **Figure 4** indicates that, if a thread is dispatched with an INVITE message, it should begin execution in `Registering`.

Our implementation of SNeF4SS loads the XML description of a servlet's contract into an object upon deployment of an application. We call this object a *static contract* because it designates only static navigation expressions, not the actual resources. When a thread starts up, it automatically invokes its negotiator (in an initialization method wrapped by `SNeF4SSSipServlet`) to initialize the thread state.

4.3 Negotiators

The negotiator object associated with a message keeps track of the state of the thread processing the message, the resources the thread is holding (and thus that it can safely access), and the resources the thread needs to acquire. A thread automatically invokes a method[11] on its negotiator on starting up to enter its initial thread state. The negotiator evaluates the conditions of the initialization rules in the static contract. It selects the target state of the first rule whose condition is true. It then queries the static contract for navigation expressions designating the resources to acquire before returning control to the servlet. If necessary, it blocks the thread while inferring and acquiring the needed resources.

A thread obtains a handle to its negotiator object by invoking the `getNegotiator` method in the `SNeF4SSSipServlet` API. To change its state, a thread invokes its negotiator's `enter` method with the name of the target state. In `enter`, the negotiator infers the resources to be released and releases them. Then, it infers and attempts to acquire any resources it does not currently hold, but needs in the target state. If some of the needed resources are unavailable, it blocks the thread. Thus, on return from `enter`, the thread can safely access these resources until it enters a new state.

Our SNeF4SS toolkit provides three alternative negotiator implementations. The first two encapsulate two standard protocols for avoiding deadlock while incrementally acquiring resources: resource numbering and gate keeper. A problem affecting throughput with both of these protocols is that, in trying to acquire a set of resources, a thread may acquire some of the needed resources and then block until other needed resources become available. This practice causes threads that need resources a blocked thread is holding to also block. In contrast, the protocol encapsulated by our third negotiator implementation avoids such needless blocking through using a more intricate two-phase locking strategy based on that used in Szumo [3, 2]. Briefly, in the Szumo protocol, a negotiator does not acquire any new resources until it is able to "claim" all the resources it needs. Then, rather than acquire the claimed resources incrementally, it acquires all of them in one atomic step. If the negotiator blocks the thread, other threads may acquire resources that the thread has claimed, in which case the negotiator needs to reclaim them when they again become available. The Szumo negotiators permit more concurrency but incur extra overhead.

Because negotiators encapsulate resource inference and a locking protocol, they can implement complex deadlock detection, recovery and prevention heuristics. For instance, the details of claiming resources in Szumo are complex because of the need to avoid deadlock and be fair—the protocol uses timestamps to prioritize contending claims for the same resource and wound-wait and backoff strategies to yield if the highest priority negotiation stalls. Implementing the Szumo protocol in applications is error prone, and beyond the abilities of many application programmers. Separating the resource inference and acquisition code from the business code makes it possible to reuse the former with different applications.

4.4 Coding a servlet

To use SNeF4SS, a servlet inherits from the appropriate derivative of `SNeF4SSServlet` and implements the business logic. Each method for processing a message obtains a handle to its negotiator in order to notify the negotiator to bind names to resources and change thread states.

Figure 5 illustrates part of the servlet code for the dating-service application. The servlet inherits from `SNeF4SSSipServlet` (line 2). A code snippet from the `doInvite` method, which a thread automatically invokes to process an INVITE message, is shown. The servlet obtains a handle to its negotiator (lines 6–7), and then objectifies and attempts to register the client user agent (lines 9–12). If registration fails, the thread sends a failure response to the client user agent and terminates (lines 10–12); otherwise, the servlet enters `Choosing` (line 14) and attempts to choose a peer (line 15). If `chooseUser` returns a peer user agent, the servlet binds a reference to the peer to a designator, `peerSS`, that is specified for `TryingToPair` state in the contract (lines 16–17) and enters `CheckingBoth` (line 18).

We elide much of this example due to space limit. The portion shown, however, illustrates the close correspondence between the state diagrams and the code. No low level synchronization code clutters the business code. Instead, the call-outs to SNeF4SS clearly demarcate state transitions.

[11] not shown in **Figure 3** as framework programmers and application programmers do not use it.

```
public class DatingSipServlet
       extends SNeF4SSSipServlet {
 ...
 void doInvite(SipServletRequest message)
 {
   SNeF4SSNegotiator ngtr =
                     getNegotiator(message);
   ...
     User client = objectify(message);

     if(!register(client)){
       message.createResponse(500).send();return;
     }

     while(true){
       ngtr.enter("Choosing");
       peer = chooseUser(client);
       if (peer != null){
         bind("peerSS", peer.getSipSession())
         ngtr.enter("CheckingBoth")
         ...
     }
   ...
}
```

Figure 5: Snippet of servlet code using SNeF4SS

This traceability makes the servlet code easier to understand, maintain and extend than code that interleaves resource inference and locking with business code.

5. SYNCHRONIZATION MODEL

To provide a semantic foundation for our SNeF4SS API, we developed a formal model of thread synchronization based on contracts. In brief, we model the business logic by a traditional state automaton. We then extend this automaton with a synchronization contract and define what it means for an execution of the automaton to satisfy the contract. This section formalizes these notions.

Definition 1. *A* business automaton *consists of*

- *a set of states, S*
- *a set of transitions, T*
- *a set of initial states, $I \subseteq S$*
- *a source mapping, $source \colon T \to S$*
- *a target mapping, $target \colon T \to S$*

In the sequel, we assume a business automaton is given. As customary, we regard the business automaton as defining a ternary *transition relation*, $\to$, and we write $s \xrightarrow{t} s'$ to mean (s, t, s') is in the transition relation, for $s, s' \in S$, $t \in T$. The transition relation formalizes how control passes from one state to another as a result of taking transitions. Then, from the transitive closure of this relation we derive an *execution relation*. For $s, s' \in S$ and $\bar{t} \in T^*$, we write $s \overset{\bar{t}}{\to}_{ex} s'$ to mean $(s, \bar{t}, s')$ is in the execution relation, and we call $\bar{t} \in T^*$ a *transition trace*. Intuitively, this assertion expresses that, starting from s, the transitions in $\bar{t}$ may be taken, in order, with the effect of leaving control in s'. In formalizing these and subsequent concepts, we write Λ for the empty sequence; $\bar{a}$ for a (finite) sequence of elements from a set A, $\bar{a} \in A^*$; $(\bar{a} \cdot a)$ for the sequence produced by appending $a \in A$ to the end of $\bar{a} \in A^*$; and, if $\bar{a} \neq \Lambda$, $last(\bar{a})$ for its last element.

Definition 2. *The transition relation, $\to$, is the smallest relation in $S \times T \times S$ such that $s \xrightarrow{t} s'$ iff $source(t) = s$ and $target(t) = s'$, for $t \in T$ and $s, s' \in S$.*

The execution relation, $\to_{ex}$, is the smallest relation in $S \times T^* \times S$ such that $s \overset{\Lambda}{\to}_{ex} s$, and $s \overset{\bar{t}}{\to}_{ex} s'$ and $s' \xrightarrow{t} s''$ implies $s \overset{(\bar{t} \cdot t)}{\to}_{ex} s''$.

The states in a business automaton represent global states, not local states of individual threads. In contrast, the states in **Figure 2** represent local states of threads. Intuitively, a global state in a business automaton for our dating service application is viewed as defining a set of active threads; associating each active thread with a thread state (i.e., a state-diagram state), a message being processed, and a SIP session and application session associated with this message; associating each SIP and application session with attributes and attribute values; and so on. Transitions of the state diagram are then viewed as inducing transitions on these global states, where we assume an interleaved model of execution. In Definition 1, we abstract away from these application-specific details. Because the business automaton does not express any information needed to synchronize threads, its executions are not synchronized in any fashion.

To enable judging if an execution is correctly synchronized, we augment a business automaton with information about threads and the resources that should be released and acquired when taking transitions.

Definition 3. *A* synchronizable business automaton *consists of*

- *a business automaton*
- *a set of threads, TH*
- *a set of resources, R*
- *a partition of T, $P \subset 2^T$*
- *a thread mapping, $m \colon P \to TH$*
- *a contract, $(c_r, c_a) \in (T \to 2^R) \times (T \to 2^R)$*

For $t \in T$, we write $[t]$ for the equivalence class in P containing t, i.e., for the set satisfying $[t] \in P$ and $t \in [t]$. The partition and thread mapping formalize that we use an interleaved model of computation—i.e., that a global transition occurs when a thread takes a local transition (e.g., a transition in a state diagram). Intuitively, $[t]$ models the local transition that produces global transition t and $m([t])$ models the thread that takes local transition $[t]$, for $t \in T$. As a notational convenience, from here on, we show the thread that takes a transition as a superscript when naming a transition. For example, we write $t^\pi \in T$ to denote a transition taken by thread π, i.e., satisfying $m([t^\pi]) = \pi$.

The contract in a synchronizable business automaton provides information about the resource requirements of threads. Namely, for a transition $t^\pi \in T$, the contract specifies the set of resources $c_r(t^\pi) \subseteq R$ that π should release and the set of resources $c_a(t^\pi) \subseteq R$ that π should acquire when it takes the transition t^π . We use this information to judge if an execution of the business automaton satisfies the contract. For the remaining discussion, assume a synchronizable business automaton is given.

We seek to define a sub-relation of the execution relation formalizing that, if threads release and acquire resources in accordance with the contract, then executing the transition trace does not violate mutual exclusion. We refer to this relation as the *synchronized execution relation*, and write $s \xrightarrow{\bar{t}}_{syex} s'$ to mean $(s, \bar{t}, s')$ belongs to the synchronized execution relation, for $s, s' \in S$ and $\bar{t} \in T^*$. The definition of this relation is complicated by wanting to express, not just that the protocol used to synchronize threads guarantees the contract is satisfied, but also that it does not "introduce" deadlocks. To achieve this latter goal, we need a view of executions that encodes, not just the order in which transitions occur, but also the points at which the resources that a thread releases become available for other threads to acquire. When a thread decides to take a transition that will release a resource, then that resource should be available for some other thread to acquire.

To provide this view, we regard a thread π as taking a local transition in two (atomic) steps: in the first step, it releases any resources in $c_r(t^\pi)$ that it holds and, in the second, it acquires all resources in $c_a(t'^\pi)$ that it does not already hold, where $[t^\pi] = [t'^\pi]$.

Definition 4. *A transition $t^\pi \in T$ induces a* release step*, denoted t^π_{rel}, and an* acquire step*, denoted t^π_{acq}. Additionally, T_{rel} denotes the set of all release steps, T_{acq} denotes the set of all acquire steps, and T_{step} denotes their union, $T_{step} = T_{rel} \cup T_{acq}$.*

As a notational convention, we write $\bar{t}_{step}$ for a sequence of steps, $\bar{t}_{step} \in T^*_{step}$.

We then "refine" the execution relation of Definition 2 to express that taking a transition involves first a release step and then an acquire step. In defining this relation, the function $proj_\pi \colon T^*_{step} \to T^*_{step}$ projects a step trace on the steps taken by thread π.

Definition 5. *The* unsynchronized step-execution relation *is the smallest relation $\to_{un} \subseteq S \times T^*_{step} \times S$, satisfying*

1. *if $s \in S$, then $s \xrightarrow{\Lambda}_{un} s$*
2. *if $s \xrightarrow{\bar{t}_{step}}_{un} s'$, $s' \xrightarrow{t^\pi} s''$, and either $proj_\pi(\bar{t}_{step}) = \Lambda$ or $last(proj_\pi(\bar{t}_{step})) \in T_{acq}$, then $s \xrightarrow{(\bar{t}_{step} \cdot t^\pi_{rel})}_{un} s'$*
3. *if $s \xrightarrow{\bar{t}_{step}}_{un} s'$, $s' \xrightarrow{t^\pi} s''$, $last(proj_\pi(\bar{t}_{step})) \in T_{rel}$, and $[t^\pi] = [last(proj_\pi(\bar{t}_{step}))]$, then $s \xrightarrow{(\bar{t}_{step} \cdot t^\pi_{acq})}_{un} s'$,*

for $s, s', s'' \in S$, $\bar{t}_{step} \in T^*_{step}$, and $t^\pi \in T$.

Here, (2) expresses that π may perform the release step for transition t^π if t^π is enabled and the last step π performed, if any, was an acquire step; and also that doing so has no effect on the state of the business automaton in which control resides. In contrast, (3) expresses that, when π performs an acquire step, control passes to the target state, and that π can perform the acquire step of a transition t^π only if the previous step it performed was a release step of a transition representing the same local transition as t^π.

To formalize the correspondence between step traces and transition traces, we introduce a function that "lifts" a step trace to a transition trace.

Definition 6. *The function $lift \colon T^*_{step} \to T^*$ satisfies $lift(\Lambda) = \Lambda$, $lift(\bar{t}_{step} \cdot t^\pi_{rel}) = lift(\bar{t}_{step})$, and $lift(\bar{t}_{step} \cdot t^\pi_{acq}) = (lift(\bar{t}_{step}) \cdot t^\pi)$, for $\bar{t}_{step} \in T^*_{step}$, $t^\pi_{rel} \in T_{rel}$, and $t^\pi_{acq} \in T_{acq}$.*

In essence, a step trace represents the trace in which a transition is viewed as taking place upon performing the transition's acquire step.

The following lemma expresses the sense in which the unsynchronized step-execution relation refines the execution relation. It is a straightforward consequence of the prior definitions.

Lemma 7. *$s \xrightarrow{\bar{t}_{step}}_{un} s'$ only if $s \xrightarrow{lift(\bar{t}_{step})}_{ex} s'$, for $s, s' \in S$ and $\bar{t}_{step} \in T^*_{step}$. Conversely, $s \xrightarrow{\bar{t}}_{ex} s'$ only if there exists a step trace, $\bar{t}_{step} \in T^*_{step}$, such that $\bar{t} = lift(\bar{t}_{step})$ and $s \xrightarrow{\bar{t}_{step}}_{un} s'$, for $s, s' \in S$ and $\bar{t} \in T^*$.*

In formalizing how a protocol should synchronize threads, we use a *holds function* to associate threads with the resources they hold—i.e., have acquired and have not yet released. We also define a function to express the set of resources that are *free* in a holds function.

Definition 8. *A holds function is a function from threads to sets of resources, $h \colon TH \to 2^R$.*

The function $free \colon (TH \to 2^R) \to 2^R$ satisfies $free(h) = R - \bigcup h(TH)$, for $h \colon TH \to 2^R$.

In essence, the resources free in a holds function are those resources that no thread holds.

We express the meanings of release and acquire steps by defining how step traces modify holds functions.

Definition 9. *The* holds relation *is the smallest relation, $\to_{ho} \subseteq (TH \to 2^R) \times T^*_{step} \times (TH \to 2^R)$, satisfying*

1. *$h \xrightarrow{\Lambda}_{ho} h$*
2. *if $h \xrightarrow{\bar{t}_{step}}_{ho} h'$, then $h \xrightarrow{(\bar{t}_{step} \cdot t^\pi_{rel})}_{ho} h'[\pi / (h'(\pi) - c_r(t^\pi))]$*
3. *if $h \xrightarrow{\bar{t}_{step}}_{ho} h'$ and $c_a(t^\pi) \subseteq free(h')$, then $h \xrightarrow{(\bar{t}_{step} \cdot t^\pi_{acq})}_{ho} h'[\pi / (h'(\pi) \cup c_a(t^\pi))]$*

Informally, (2) expresses that performing the release step of t^π frees any resources in $c_r(t^\pi)$ that π holds, and (3) expresses that π can take the acquire step of t^π only if no other threads hold any resources in $c_a(t^\pi)$ and that it holds these resources upon doing so.

Composing the unsynchronized step-execution relation with the holds relation produces the *synchronized step-execution relation*, which defines what it means for a step trace to be synchronized.

DEFINITION 10. *The synchronized step-execution relation is the relation*

$$\underset{sy}{\rightarrow} \subseteq (TH \rightarrow 2^R) \times S \times T^*_{step} \times (TH \rightarrow 2^R) \times S$$

satisfying $(h, s) \xrightarrow{\bar{t}_{step}}_{sy} (h', s')$ *iff* $h \xrightarrow{\bar{t}_{step}}_{ho} h'$ *and* $s \xrightarrow{\bar{t}_{step}}_{un} s'$.

We then define the synchronized execution relation by lifting synchronized step traces and hiding the holds functions. We also require that, before starting execution, the state and holds function are properly initialized.

DEFINITION 11. *The synchronized execution relation,*

$$\underset{syex}{\rightarrow} \subseteq S \times T^* \times S,$$

satisfies $s \xrightarrow{\bar{t}}_{syex} s'$ *iff* $s \in I$ *and there exists a sequence of steps* $\bar{t}_{step} \in T^*_{step}$, *a holds function* $h\colon TH \rightarrow 2^R$, *and a state* $s \in S$ *such that* $\bar{t} = lift(\bar{t}_{step})$ *and* $(\emptyset, s) \xrightarrow{\bar{t}_{step}}_{sy} h, s'$.

Thus, a synchronizable business automaton determines a synchronized execution relation expressing that, starting from an initial state, a transition trace can be executed while satisfying the contract.

This definition captures two fundamental properties of any implementation of SNeF4SS. First and foremost, it expresses that negotiators must synchronize threads so that the contract is satisfied. Thus, on entering a state, a thread can assume it holds all and only the resources that the contract stipulates it needs, and that it retains exclusion on these resources until it leaves the state. Second, this definition expresses that negotiators must not "introduce" deadlock. This property follows from the atomicity of the acquisition step. Modeling acquisition as an atomic step requires that a negotiator does not hold any resource it needs to acquire indefinitely unless it eventually acquires all of them (at which point the entire set of needed resources is viewed as having been "atomically" acquired).

6. RELATED WORK

Our work capitalizes on separating synchronization logic from business logic in order to facilitate programming of SIP applications. Declarative synchronization contracts are what make separation possible. We are not aware of research that addresses synchronization problems specifically in the SIP domain. Frameworks and methodologies have been developed, however, to facilitate handling of the complexity caused by SIP and other communication protocols. This section overviews related work on separating synchronization concerns from business concerns in programming general concurrent systems, and then briefly discusses a couple of recent approaches to raise the level of abstraction for SIP programming.

Separation of concerns is a general principle in software engineering for handling complexity in applications [15]. Our abstract synchronization model is motivated by Meyer's *Design-by-Contract* principle [16] to facilitate separation of the synchronization concern. In this principle, contracts describe the rights and responsibilities between operation consumers and suppliers. In our approach, contracts describe the rights and responsibilities between SIP applications and negotiators.

Our language for specifying contracts can be viewed as generalizing the notion of *critical regions* [9, 6], one of the earlier language mechanism proposed for declaratively specifying the resource needs of threads. Entering a thread state is similar to entering a critical region. In contrast to a critical region, however, our contract specifications can indicate multiple resources that should be acquired "atomically" before entering a thread state. Moreover, a thread state may have multiple entry and exit points, whereas a critical region is a simple block statement.

An important class of more recent approaches use *region invariants* to declare synchronization constraints within designated regions of the code (e.g., constaining the numbers of threads that can be executing concurrently). Region invariants may be declared separately from the code, as in SyncGen [5], or given by annotating the code directly, as in JAC [8]. Prior to compilation, these approaches generate synchronization code from the region invariants and weave it into a subject program. By contrast, our approach preserves the structure of the original program and relegates thread synchronization to an independent middleware.

The *concurrency controllers* design pattern provides yet another approach to separating synchronization concerns [4]. Using this pattern, a programmer writes a concurrency controller, which controls access to shared objects, much like our negotiators. Instead of explicitly acquiring shared objects, threads inform a centralized concurrency controller at runtime of actions involving shared objects. The concurrency controller dynamically triggers or postpones the action to enforce mutual exclusion. Intuitively, these actions collectively encapsulate a synchronization policy. While similar in this respect to our negotiators, a concurrency controller is a centralized mechanism associated with all resources that need protection, not with threads.

To facilitate handling of the complexity caused by SIP and other communication protocols, some recent research proposes innovative programming frameworks and methodologies. In [12], a programmer writes applications on top of a programming framework that provides programming support for manipulating interactions among communication entities. The programmer declaratively specifies the interaction characteristics of these entities to automatically generate the framework. In [17], by contrast, a programmer describes a SIP application as a composition of finite state machines, called *E-charts*. Actions associated with the states and transitions of E-charts express the message-processing logic. A compiler translates the E-charts into servlet classes. These frameworks and methodologies raise the level of abstraction for programming SIP applications. None, however, deals with problems of synchronization in SIP containers.

7. CONCLUSION AND FUTURE WORK

Programming thread synchronization complicates the development of reliable and efficient SIP applications deployed to SIP servlet containers. We proposed a novel model that facilitates isolation of this complication. By automating the obligation of thread synchronization in an independent middleware, our approach promotes correctness and flexibility. In addition, the business code written by application programmers is also highly traceable to high level state models, which aids comprehension, maintenance, and extension.

In our future work, we will integrate our approach with state-machine based programming paradigms, such as Statemate [7] and ECharts [17]. Once integrated, a programmer could perform a model-checking process to verify whether a generated state-model, in terms of the semantics introduced in **Section 5**, agrees its synchronization contract or whether the model under the contract could cause synchronization failures. Additionally, the application translated from this model could automatically include the code that instantiates our approach, such as announcing transitions.

Our future work also involves promotion of our approach to better serve realistic SIP applications. For instance, our model could be generalized to support *conditional resources* by permiting a resource designator to be associated with a boolean expression that signifies the condition under which a thread needs the corresponding resource. Conditional resources would facilitate specification of contracts for applications that use `wait` and `signal` synchronization primitives [9]. In addition, our framework could also be extended to support *vertical composition* of synchronization contracts. Currently, a programmer must explicitly declare the entire set of resources that a thread (directly or indirectly) needs in a global contract. For better modularity, a global contract could be obtained by vertically composing local contracts each specifying what other resources are needed to perform operations that the resource provides.

Acknowledgements: Partial support was provided for this research by NSF grant CCF 0702667, LogicBlox Inc., and AT&T Research Laboratory.

The authors also thank G. Bond, H. Purdy, T. Smith, V. Subramonian, and P. Zave for indispensable explanations, feedback and guidance.

8. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Journal on Software Testing, Verification and Reliability (STVR)*, 2003.

[2] R. Behrends. *Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages.* PhD thesis, Michigan State University, East Lansing, Michigan USA, Dec. 2003.

[3] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of FSE'2000*, 2000.

[4] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. of the IEEE International Conference on Automated Software Enginerring*, 2004.

[5] X. Deng et al. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. of the IEEE International Conference on Software Engineering (ICSE'02)*, 2002.

[6] P. B. Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *Software Engineering, IEEE Transactions on*, 16(4):403–414, Apr 1990.

[8] M. Haustein and K.-P. Löhr. Jac: declarative java concurrency: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(5):519–546, 2006.

[9] C. A. R. Hoare and R. H. Perrott Ed. *Towards a theory of parallel programming.* London: Academic, 1972.

[10] Y. Huang, L. K. Dillon, and R. E. K. Stirewalt. On mechanisms for deadlock avoidance in SIP servlet containers. In H. Schulzrinne, R. State, and S. Niccolini, editors, *IPTComm*, volume 5310 of *Lecture Notes in Computer Science*, pages 196–216. Springer, 2008.

[11] Y. Huang, L. K. Dillon, and R. E. K. Stirewalt. Prototyping synchronization policies for existing programs. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009*, 2009.

[12] W. Jouve, N. Palix, C. Consel, and P. Kadionik. A SIP-based programming framework for advanced telephony applications. pages 1–20, 2008.

[13] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008.

[14] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[15] T. Mens and M. Wermelinger. Separation of concerns for software evolution. *Journal of Software Maintenance*, 14(5):311–315, 2002.

[16] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10), 1992.

[17] T. M. Smith and G. W. Bond. ECharts for SIP servlets: a state-machine programming environment for voip applications. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 89–98, New York, NY, USA, 2007. ACM.

[18] J. Wilkiewicz and M. Kulkarni. JSR 289: SIP servlet specification v1.1.